Formalization of an Algorithm for Greedily Computing Associative Aggregations on Sliding Windows

Lukas Heimes Dmitriy Traytel Joshua Schneider

February 6, 2026

Abstract

Basin et al.'s sliding window algorithm (SWA) [1] is an algorithm for combining the elements of subsequences of a sequence with an associative operator. It is greedy and minimizes the number of operator applications. We formalize the algorithm and verify its functional correctness. We extend the algorithm with additional operations and provide an alternative interface to the slide operation that does not require the entire input sequence.

Contents

1 Sliding Window Algorithm **1**

2 Correctness **4**
2.1 Correctness of the Slide Function 5
2.2 Correctness of the Sliding Window Algorithm 7
2.3 Summary of the Correctness Proof 7

3 Alternative Slide Interface and Additional Operations **7**
3.1 Alternative Slide Interface 7
3.2 Updating all Values in the Tree 8
3.3 Updating the Rightmost Leaf of the Tree 9

1 Sliding Window Algorithm

datatype *'a tree* =
  Leaf
| *Node* (*l*: *nat*) (*r*: *nat*) (*val*: *'a option*) (*lchild*: *'a tree*) (*rchild*: *'a tree*)
where
  l Leaf = *0*
| *r Leaf* = *0*
| *val Leaf* = *None*
| *lchild Leaf* = *Leaf*
| *rchild Leaf* = *Leaf*

lemma *neq_Leaf_if_l_gt0*: *0* < *l t* ⟹ *t* ≠ *Leaf*
  ⟨*proof*⟩

primrec *discharge* :: *'a tree* ⇒ *'a tree* **where**
  discharge Leaf = *Leaf*
| *discharge* (*Node i j* _ *t u*) = *Node i j None t u*

instantiation *option* :: (*semigroup_add*) *semigroup_add* **begin**

fun *plus_option* :: *'a option* ⇒ *'a option* ⇒ *'a option* **where**
  plus_option None _ = None
| *plus_option _ None = None*
| *plus_option (Some a) (Some b) = Some (a + b)*

instance ⟨*proof*⟩

end

fun *combine* :: *'a* :: *semigroup_add tree* ⇒ *'a tree* ⇒ *'a tree* **where**
  combine t Leaf = t
| *combine Leaf t = t*
| *combine t u = Node (l t) (r u) (val t + val u) (discharge t) u*

lemma *combine_non_Leaves*: ⟦*t ≠ Leaf*; *u ≠ Leaf*⟧ ⟹ *combine t u = Node (l t) (r u) (val t + val u) (discharge t) u*
⟨*proof*⟩

lemma *r_combine_non_Leaves*: ⟦*t ≠ Leaf*; *u ≠ Leaf*⟧ ⟹ *r (combine t u) = r u*
  ⟨*proof*⟩

type_synonym *window* = *nat* × *nat*

definition *window* :: *'a list* ⇒ *window* ⇒ *bool* **where**
  window as = (λ(*l*, *r*). *0 < l* ∧ *l* ≤ *r* ∧ *r* ≤ *length as*)

definition *windows* :: *'a list* ⇒ *window list* ⇒ *bool* **where**
  windows as ws = ((∀ *w* ∈ *set ws*. *window as w*) ∧
    sorted (*map fst ws*) ∧ *sorted* (*map snd ws*))

function *reusables* :: *'a tree* ⇒ *window* ⇒ *'a tree list* **where**
  reusables t w = (*if fst w > r t then* [] *else if fst w = l t then* [*t*]
    else let v = lchild t; *u = rchild t in if fst w ≥ l u then*
      reusables u w else u # reusables v w)
  ⟨*proof*⟩
termination
  ⟨*proof*⟩

declare *reusables.simps*[*simp del*]

lemma *reusables_Leaf*[*simp*]: *0 < fst w* ⟹ *reusables Leaf w* = []
  ⟨*proof*⟩

primrec *well_shaped* :: *'a tree* ⇒ *bool* **where**
  well_shaped Leaf = True
| *well_shaped (Node i j _ t u) = (i ≤ j* ∧ (*i = j* ⟶ *t = Leaf* ∧ *u = Leaf*) ∧
    (*i < j* ⟶ *t ≠ Leaf* ∧ *u ≠ Leaf* ∧ *well_shaped t* ∧ *well_shaped u* ∧
      i = l t ∧ *j = r u* ∧ *Suc (r t) = l u*))

lemma *l_lchild_eq_l_if_well_shaped*[*simp*]:
  ⟦*well_shaped t*; *l t < r t*⟧ ⟹ *l (lchild t) = l t*
  ⟨*proof*⟩

lemma *r_rchild_eq_r_if_well_shaped*[*simp*]:
  ⟦*well_shaped t*; *l t < r t*⟧ ⟹ *r (rchild t) = r t*
  ⟨*proof*⟩

lemma *r_lchild_eq_l_rchild_if_well_shaped*:
⟦*well_shaped t*; *l t* < *r t*⟧ ⟹ *r* (*lchild t*) = *l* (*rchild t*) − *1*
⟨*proof*⟩

lemma *r_lchild_le_r*: *well_shaped t* ⟹ *r* (*lchild t*) ≤ *r t*
⟨*proof*⟩

lemma *well_shaped_lchild*[*simp*]: *well_shaped t* ⟹ *well_shaped* (*lchild t*)
⟨*proof*⟩

lemma *well_shaped_rchild*[*simp*]: *well_shaped t* ⟹ *well_shaped* (*rchild t*)
⟨*proof*⟩

definition *adjacent* **where**
adjacent w ts = (*Leaf* ∉ *set ts* ∧
list_all2 (λ*t u*. *l t* = *Suc* (*r u*)) (*butlast ts*) (*tl ts*) ∧
(*ts* = [] ∨ (*l* (*last ts*) = *fst w* ∧ *r* (*hd ts*) = *snd w*)))

lemma *adjacent_Nil*[*simp*]: *adjacent w* []
⟨*proof*⟩

lemma *adjacent_Cons*: *adjacent w* (*t* # *ts*) =
(*t* ≠ *Leaf* ∧ *r t* = *snd w* ∧ (*case ts of* [] ⇒ *l t* = *fst w*
| *u* # *us* ⇒ *adjacent* (*fst w*, *r u*) *ts* ∧ *l t* = *Suc* (*r u*)))
⟨*proof*⟩

lemma *adjacent_ConsI*: ⟦*t* ≠ *Leaf*; *r t* = *snd w*;
(*case ts of* [] ⇒ *l t* = *fst w*
| *u* # *us* ⇒ *adjacent* (*fst w*, *r u*) *ts* ∧ *l t* = *Suc* (*r u*))⟧ ⟹
adjacent w (*t* # *ts*)
⟨*proof*⟩

lemma *adjacent_singleton*: *t* ≠ *Leaf* ⟹ *adjacent* (*l t*, *r t*) [*t*]
⟨*proof*⟩

lemma *append_Cons_eq_append_append*: *xs* @ *y* # *ys* = *xs* @ [*y*] @ *ys*
⟨*proof*⟩

lemma *list_all2_append_singletonI*: ⟦*list_all2 P xs ys*; *P x y*⟧ ⟹ *list_all2 P* (*xs* @ [*x*]) (*ys* @ [*y*])
⟨*proof*⟩

lemma *list_all2_Cons_append_singletonI*: ⟦*xs* ≠ []; *list_all2 P* (*x* # *butlast xs*) *ys*; *P* (*last xs*) *y*⟧ ⟹
list_all2 P (*x* # *xs*) (*ys* @ [*y*])
⟨*proof*⟩

lemma *adjacent_appendI*: ⟦*0* < *fst w*; *fst w* ≤ *snd w*;
(*case us of* [] ⇒ *adjacent w ts*
| *u* # *us*′ ⇒ *adjacent* (*Suc* (*r u*), *snd w*) *ts* ∧ *adjacent* (*fst w*, (*case ts of* [] ⇒ *snd w* | *ts* ⇒ *r u*)) (*u* # *us*′))⟧ ⟹
adjacent w (*ts* @ *us*)
⟨*proof*⟩

lemma *adjacent_Cons_implies_adjacent*: *adjacent* (*a*, *b*) (*t* # *ts*) ⟹ *adjacent* (*a*, *l t* − *Suc 0*) *ts*
⟨*proof*⟩

lemma (**in** *semigroup_add*) *fold_add_add*: *fold* (+) *xs* (*x* + *y*) = *fold* (+) *xs x* + *y*
⟨*proof*⟩

context
  fixes *as* :: *'a* :: *semigroup_add list*
  and *ws* :: *window list*
begin

abbreviation *atomic* **where**
  atomic i ≡ *Node i i* (*Some* (*nth as* (*i* − *1*))) *Leaf Leaf*

definition *atomics* :: *nat* ⇒ *nat* ⇒ *'a tree list* **where**
  atomics i j ≡ *map atomic* (*rev* [*i* ..< *Suc j*])

definition *slide* :: *'a tree* ⇒ *window* ⇒ *'a tree* **where**
  slide t w =
    (*let*
      ts = *atomics* (*max* (*fst w*) (*Suc* (*r t*))) (*snd w*);
      ts' = *reusables t w*
    in fold combine (*ts* @ *ts'*) *Leaf*)

primrec *iterate* :: *'a tree* ⇒ *window list* ⇒ *'a list* **where**
  iterate t [] = []
| *iterate t* (*w* # *xs*) = (*let t'* = *slide t w in the* (*val t'*) # *iterate t' xs*)

definition *sliding_window* :: *'a list* **where**
  sliding_window = *iterate Leaf ws*

2 Correctness

abbreviation *sum* **where**
  sum i j ≡ *fold* (+) (*rev* (*map* (*nth as*) [*i* − *1* ..< *j* − *1*])) (*nth as* (*j* − *1*))

primrec *well_valued0* :: *'a tree* ⇒ *bool* **where**
  well_valued0 Leaf = *True*
| *well_valued0* (*Node i j a t u*) = (*0* < *i* ∧ *j* ≤ *length as* ∧ (*a* ≠ *None* ⟶ *a* = *Some* (*sum i j*)) ∧
    well_valued0 t ∧ *well_valued0 u* ∧ (*u* = *Leaf* ∨ *val u* ≠ *None*))

abbreviation *well_valued* :: *'a tree* ⇒ *bool* **where**
  well_valued t ≡ (*well_valued0 t* ∧ (*t* ≠ *Leaf* ⟶ *val t* ≠ *None*))

definition *valid* :: *'a tree* ⇒ *bool* **where**
  valid t = (*well_shaped t* ∧ *well_valued t*)

lemma *valid_Leaf*: *valid Leaf*
  ⟨*proof*⟩

lemma *add_sum*:
  assumes *i* > *0 j* ≥ *i k* > *j*
  shows *sum i j* + *sum* (*Suc j*) *k* = *sum i k*
⟨*proof*⟩

lemma *well_valued0_rchild_if_well_valued0*[*simp*]: *well_valued0 t* ⟹ *well_valued0* (*rchild t*)
  ⟨*proof*⟩

lemma *well_valued0_lchild_if_well_valued0*[*simp*]: *well_valued0 t* ⟹ *well_valued0* (*lchild t*)
  ⟨*proof*⟩

lemma *valid_rchild_if_valid*: *valid t* ⟹ *valid* (*rchild t*)
  ⟨*proof*⟩

lemma *val_eq_Some_sum_if_valid_neq_Leaf*: $[\![valid\ t;\ t \neq Leaf]\!] \Longrightarrow val\ t = Some\ (sum\ (l\ t)\ (r\ t))$
⟨*proof*⟩

2.1 Correctness of the Slide Function

lemma *adjacent_atomics*: *adjacent* (i, j) $(atomics\ i\ j)$
⟨*proof*⟩

lemma *valid_atomics*: $[\![t \in set\ (atomics\ i\ j);\ 0 < i;\ j \leq length\ as]\!] \Longrightarrow valid\ t$
⟨*proof*⟩

lemma *reusables_neq_Nil_if_well_shaped_and_overlapping*:
$[\![well_shaped\ t;\ l\ t \leq fst\ w;\ r\ t \leq snd\ w;\ fst\ w \leq r\ t]\!] \Longrightarrow reusables\ t\ w \neq []$
⟨*proof*⟩

lemma *reusables_lchild_neq_Nil_under_some_conditions*:
$[\![well_shaped\ t;\ l\ t \leq fst\ w;\ r\ t \leq snd\ w;\ fst\ w \neq l\ t;\ r\ t \geq fst\ w;\ l\ (rchild\ t) > fst\ w]\!] \Longrightarrow$
$reusables\ (lchild\ t)\ w \neq []$
⟨*proof*⟩

lemma *adjacent_reusables*: $[\![0 < fst\ w;\ well_shaped\ t;\ l\ t \leq fst\ w;\ r\ t \leq snd\ w]\!] \Longrightarrow$
$adjacent\ (fst\ w,\ r\ t)\ (reusables\ t\ w)$
⟨*proof*⟩

lemma *valid_rchild_if_well_valued0*: $[\![well_shaped\ t;\ well_valued0\ t]\!] \Longrightarrow valid\ (rchild\ t)$
⟨*proof*⟩

lemma *valid_reusables_under_some_conditions*:
$[\![0 < fst\ w;\ well_valued0\ t;\ well_shaped\ t;\ l\ t < fst\ w;\ r\ t \leq snd\ w]\!] \Longrightarrow$
$\forall\, t' \in set\ (reusables\ t\ w).\ valid\ t'$
⟨*proof*⟩

lemma *valid_reusables*:
assumes $0 < fst\ w\ valid\ t\ l\ t \leq fst\ w\ r\ t \leq snd\ w$
shows $\forall\, t' \in set\ (reusables\ t\ w).\ valid\ t'$
⟨*proof*⟩

lemma *combine_valid_Nodes_aux*:
assumes *prems*: $0 < l\ a\ a \neq Leaf\ z \neq Leaf\ l\ z = Suc\ (r\ a)\ well_shaped\ a\ well_shaped\ z$
$well_valued0\ a\ val\ a = Some\ va\ well_valued0\ z\ val\ z = Some\ vz$
shows $va + vz = fold\ (+)\ (rev\ (map\ ((!)\ as)\ [l\ a - Suc\ 0..<r\ z - Suc\ 0]))\ (as\ !\ (r\ z - Suc\ 0))$
⟨*proof*⟩

lemma *discharge_is_Leaf*[*simp*]: $discharge\ a = Leaf \longleftrightarrow a = Leaf$
⟨*proof*⟩

lemma *well_shaped_discharge*[*simp*]: $well_shaped\ a \Longrightarrow well_shaped\ (discharge\ a)$
⟨*proof*⟩

lemma *well_valued0_discharge*[*simp*]: $well_valued0\ a \Longrightarrow well_valued0\ (discharge\ a)$
⟨*proof*⟩

lemma *l_discharge*[*simp*]: $l\ (discharge\ a) = l\ a$
⟨*proof*⟩

lemma *r_discharge*[*simp*]: *r* (*discharge a*) = *r a*
⟨*proof*⟩

lemma *well_shaped_lr*: *well_shaped a* ⟹ *l a* ≤ *r a*
⟨*proof*⟩

lemma *well_valued0_r*: *well_valued0 a* ⟹ *a* ≠ *Leaf* ⟹ *r a* ≤ *length as*
⟨*proof*⟩

lemma *valid_combine_if_valid*: ⟦*0* < *l a*; *valid a*; *valid z*; *a* ≠ *Leaf*; *z* ≠ *Leaf*; *l z* = *Suc* (*r a*)⟧ ⟹
valid (*combine a z*)
⟨*proof*⟩

lemma *combine_neq_Leaf_if_both_non_Leaf*: ⟦*a* ≠ *Leaf*; *z* ≠ *Leaf*⟧ ⟹
combine a z ≠ *Leaf*
⟨*proof*⟩

lemma *valid_fold_combine*: ⟦*0* < *fst w*; *ts* = *h* # *ts'*; ∀ *t* ∈ *set ts*. *valid t*; *adjacent* (*fst w*, *l h* − *1*) *ts'*;
valid z; *z* ≠ *Leaf*; *l z* = (*case ts' of* [] ⇒ *fst w* | *t*₁ # *ts''* ⇒ *Suc* (*r t*₁)); *r z* = *snd w*⟧ ⟹
valid (*fold combine ts' z*) ∧
l (*fold combine ts' z*) = *fst w* ∧ *r* (*fold combine ts' z*) = *snd w*
⟨*proof*⟩

lemma *valid_fold_combine_Leaf*:
assumes *0* < *fst w ts* = *h* # *ts'* ∀ *t* ∈ *set ts*. *valid t adjacent w ts*
shows *valid* (*fold combine ts Leaf*) ∧
l (*fold combine ts Leaf*) = *fst w* ∧ *r* (*fold combine ts Leaf*) = *snd w*
⟨*proof*⟩

lemma *adjacent_atomics_nonempty_reusables*:
fixes *x* :: *'a tree* **and** *xs* :: *'a tree list*
assumes *a1*: *0* < *fst w*
and *a2*: *l t* ≤ *fst w*
and *a3*: *r t* ≤ *snd w*
and *a4*: *valid t*
and *a5*: *reusables t w* = *x* # *xs*
shows *adjacent* (*Suc* (*r x*), *snd w*) (*atomics* (*max* (*fst w*) (*Suc* (*r t*))) (*snd w*))
⟨*proof*⟩

lemma *adjacent_Cons_r*: *adjacent* (*a*, *r t*) (*x* # *xs*) ⟹ *adjacent* (*a*, *r x*) (*x* # *xs*)
⟨*proof*⟩

lemma *adjacent_Cons_r2*:
adjacent (*fst w*, *r t*) (*x* # *xs*) ⟹ *0* < *fst w* ⟹ *fst w* ≤ *snd w* ⟹ *r t* ≤ *snd w* ⟹
atomics (*max* (*fst w*) (*Suc* (*r t*))) (*snd w*) = [] ⟹
adjacent w (*x* # *xs*)
⟨*proof*⟩

lemma *adjacent_append_atomics_reusables*:
⟦*0* < *fst w*; *fst w* ≤ *snd w*; *valid t*; *l t* ≤ *fst w*; *r t* ≤ *snd w*⟧ ⟹
adjacent w (*atomics* (*max* (*fst w*) (*Suc* (*r t*))) (*snd w*) @ *reusables t w*)
⟨*proof*⟩

lemma *valid_append_atomics_reusables*: ⟦*0* < *fst w*; *valid t*; *l t* ≤ *fst w*; *r t* ≤ *snd w*; *snd w* ≤ *length as*⟧ ⟹

$\forall t \in set\ (atomics\ (max\ (fst\ w)\ (Suc\ (r\ t)))\ (snd\ w)\ @\ reusables\ t\ w).\ valid\ t$
⟨*proof*⟩

lemma *append_atomics_reusables_neq_Nil*: ⟦$0 < fst\ w$; $fst\ w \leq snd\ w$; *valid t*; $l\ t \leq fst\ w$; $r\ t \leq snd\ w$⟧ ⟹
$atomics\ (max\ (fst\ w)\ (Suc\ (r\ t)))\ (snd\ w)\ @\ reusables\ t\ w \neq []$
⟨*proof*⟩

lemma *valid_slide*:
assumes $0 < fst\ w\ fst\ w \leq snd\ w$ *valid t* $l\ t \leq fst\ w\ r\ t \leq snd\ w\ snd\ w \leq length\ as$
shows $valid\ (slide\ t\ w) \wedge l\ (slide\ t\ w) = fst\ w \wedge r\ (slide\ t\ w) = snd\ w$
⟨*proof*⟩

2.2 Correctness of the Sliding Window Algorithm

lemma *iterate_eq_map_sum*: ⟦*valid t*; *windows as xs*; (*case xs of* [] ⇒ *True* | $x\ \#\ xs' \Rightarrow l\ t \leq fst\ x \wedge r\ t \leq snd\ x$)⟧ ⟹
$iterate\ t\ xs = map\ (\lambda w.\ sum\ (fst\ w)\ (snd\ w))\ xs$
⟨*proof*⟩

theorem *correctness*: *windows as ws* ⟹ $sliding_window = map\ (\lambda w.\ sum\ (fst\ w)\ (snd\ w))\ ws$
⟨*proof*⟩

end

2.3 Summary of the Correctness Proof

We closely follow Basin et al.'s proof outline [1].

1. Lemma 1, the correctness result about the function *slide*, is formalized by *SWA.valid_slide*. It follows from the following auxiliary facts:
   - Fact (a) is formalized by *SWA.adjacent_reusables* and *SWA.valid_reusables*.
   - Fact (b) is formalized by *SWA.adjacent_atomics* and *SWA.valid_atomics*.
   - Fact (c) is formalized by *SWA.valid_fold_combine_Leaf*.
2. Theorem 2, the correctness result about the function *sliding_window*, is formalized by *SWA.correctness*.

3 Alternative Slide Interface and Additional Operations

3.1 Alternative Slide Interface

The slide operation above takes the *entire* input sequence as a parameter. This is often impractical. We provide an alternative interface to the slide operation that takes only the *new* elements as a parameter.

abbreviation *atomic′* **where**
$atomic'\ as\ b\ idx \equiv Node\ b\ b\ (Some\ (nth\ as\ idx))\ Leaf\ Leaf$

abbreviation *atomics′* :: *'a list* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *'a tree list* **where**
$atomics'\ as\ i\ j\ sidx \equiv map\ (\lambda b.\ atomic'\ as\ b\ (b - sidx))\ (rev\ [i\ ..<\ Suc\ j])$

definition *slide'* :: *'a* :: *semigroup_add list* ⇒ *'a tree* ⇒ *window* ⇒ *'a tree* **where**
slide' as t w =
(*let*
ts = *atomics' as* (*max* (*fst w*) (*Suc* (*r t*))) (*snd w*) (*Suc* (*r t*));
ts' = *reusables t w*
in fold combine (*ts* @ *ts'*) *Leaf*)

lemma *slide_eq_slide'*:
assumes *0* < *fst w fst w* ≤ *snd w valid as t r t* = *length as l t* ≤ *fst w r t* ≤ *snd w snd w* ≤ *length* (*as* @ *as'*)
shows *slide* (*as* @ *as'*) *t w* = *slide' as' t w*
⟨*proof*⟩

lemma *sum_eq_sum_append*: ⟦*0* < *i*; *i* ≤ *j*; *j* ≤ *length as*⟧ ⟹ *sum as i j* = *sum* (*as* @ *as'*) *i j*
⟨*proof*⟩

lemma *well_valued0_append*: ⟦*well_shaped t*; *well_valued0 as t*⟧ ⟹ *well_valued0* (*as* @ *as'*) *t*
⟨*proof*⟩

lemma *valid_append*: *valid as t* ⟹ *valid* (*as* @ *as'*) *t*
⟨*proof*⟩

lemma *valid_slide_append*: ⟦*0* < *fst w*; *fst w* ≤ *snd w*; *valid as t*; *l t* ≤ *fst w*; *r t* ≤ *snd w*; *snd w* ≤ *length as* + *length as'*⟧ ⟹
valid (*as* @ *as'*) (*slide* (*as* @ *as'*) *t w*) ∧ *l* (*slide* (*as* @ *as'*) *t w*) = *fst w* ∧ *r* (*slide* (*as* @ *as'*) *t w*) = *snd w*
⟨*proof*⟩

theorem *valid_slide'*:
assumes *0* < *fst w fst w* ≤ *snd w valid as t length as* = *r t length as'* ≥ *snd w* − *r t l t* ≤ *fst w r t* ≤ *snd w*
shows *valid* (*as* @ *as'*) (*slide' as' t w*) ∧ *l* (*slide' as' t w*) = *fst w* ∧ *r* (*slide' as' t w*) = *snd w*
⟨*proof*⟩

3.2 Updating all Values in the Tree

So far, we have assumed that the sequence is fixed. However, under certain conditions, SWA can be applied even if the sequence changes. In particular, if a function that distributes over the associative operation is mapped onto the sequence, validity of the tree can be preserved by mapping the same function onto the tree using *map_tree*.

lemma *map_tree_eq_Leaf_iff*: *map_tree f t* = *Leaf* ⟷ *t* = *Leaf*
⟨*proof*⟩

lemma *l_map_tree_eq_l*[*simp*]: *l* (*map_tree f t*) = *l t*
⟨*proof*⟩

lemma *r_map_tree_eq_r*[*simp*]: *r* (*map_tree f t*) = *r t*
⟨*proof*⟩

lemma *val_map_tree_neq_None*: *val t* ≠ *None* ⟹ *val* (*map_tree f t*) ≠ *None*
⟨*proof*⟩

lemma *well_shaped_map_tree*: *well_shaped t* ⟹ *well_shaped* (*map_tree f t*)
⟨*proof*⟩

lemma *fold_distr*: (∀ *x y*. *f* (*x* + *y*) = *f x* + *f y*) ⟹ *f* (*fold* (+) *list e*) = *fold* (+) (*map f list*) (*f e*)

⟨*proof*⟩

lemma *map_rev_map_nth_eq*: ∀ *x* ∈ *set xs*. *x* < *length as* ⟹ *map f* (*rev* (*map* ((!) *as*) *xs*)) = *rev* (*map* ((!) (*map f as*)) *xs*)
⟨*proof*⟩

lemma *f_nth_eq_map_f_nth*: ⟦*as* ≠ []; *length as* ≥ *n*⟧ ⟹ *f* (*as* ! (*n* − *Suc 0*)) = *map f as* ! (*n* − *Suc 0*)
⟨*proof*⟩

lemma *well_valued0_map_map_tree*:
⟦∀ *x y*. *f* (*x* + *y*) = *f x* + *f y*; *well_shaped t*; *well_valued0 as t*; *r t* ≤ *length as*; *as* ≠ []⟧ ⟹
well_shaped (*map_tree f t*) ∧ *well_valued0* (*map f as*) (*map_tree f t*)
⟨*proof*⟩

lemma *valid_map_map_tree*:
assumes ∀ *x y*. *f* (*x* + *y*) = *f x* + *f y valid as t r t* ≤ *length as*
shows *valid* (*map f as*) (*map_tree f t*)
⟨*proof*⟩

lemma *valid_Nil_iff*: *valid* [] *t* ⟷ *t* = *Leaf*
⟨*proof*⟩

3.3 Updating the Rightmost Leaf of the Tree

We provide a function to update the rightmost leaf of the tree. This may be used in an online setting where the input sequence is not known in advance to update the latest observed element using the same associative operation used in SWA. We show that validity of the tree is preserved in this case.

fun *update_rightmost* :: (′*a* ⇒ ′*a*) ⇒ ′*a tree* ⇒ ′*a tree* **where**
update_rightmost _ *Leaf* = *Leaf*
| *update_rightmost f* (*Node i j a t u*) = *Node i j* (*map_option f a*) *t* (*update_rightmost f u*)

lemma *update_rightmost_eq_Leaf_iff*: *update_rightmost f t* = *Leaf* ⟷ *t* = *Leaf*
⟨*proof*⟩

lemma *l_update_rightmost_eq_l*[*simp*]: *l* (*update_rightmost f t*) = *l t*
⟨*proof*⟩

lemma *r_update_rightmost_eq_r*[*simp*]: *r* (*update_rightmost f t*) = *r t*
⟨*proof*⟩

lemma *val_update_rightmost_neq_None*: *val t* ≠ *None* ⟹ *val* (*update_rightmost f t*) ≠ *None*
⟨*proof*⟩

lemma *well_shaped_update_rightmost*: *well_shaped t* ⟹ *well_shaped* (*update_rightmost f t*)
⟨*proof*⟩

lemma *sum_eq_sum_prepend*: ⟦0 < *i*; *i* ≤ *j*; *length xs* < *i*; *length ys* = *length xs*⟧ ⟹ *sum* (*xs* @ *as*) *i j* = *sum* (*ys* @ *as*) *i j*
⟨*proof*⟩

lemma *well_valued0_prepend*: ⟦*length xs* ≤ *l t* − *1*; *length ys* = *length xs*; *well_shaped t*; *well_valued0* (*xs* @ *as*) *t*⟧ ⟹ *well_valued0* (*ys* @ *as*) *t*
⟨*proof*⟩

lemma *valid_prepend*: ⟦*length xs* ≤ *l t* − *1*; *length ys* = *length xs*; *valid* (*xs* @ *as*) *t*⟧ ⟹ *valid* (*ys* @ *as*) *t*

⟨*proof*⟩

lemma *take_eq_append_take_take_drop*: $m \leq n \Longrightarrow$ *take n xs* = *take m xs* @ *take* (*n*−*m*) (*drop m xs*)
⟨*proof*⟩

lemma *well_valued0_take_r*: ⟦*well_shaped t*; *well_valued0 as t*⟧ ⟹ *well_valued0* (*take* (*r t*) *as*) *t*
⟨*proof*⟩

lemma *valid_take_r*: *valid as t* ⟹ *valid* (*take* (*r t*) *as*) *t*
⟨*proof*⟩

lemma *well_valued0_butlast*: ⟦*well_shaped t*; *well_valued0 as t*; *r t* < *length as*⟧ ⟹ *well_valued0* (*butlast as*) *t*
⟨*proof*⟩

lemma *well_valued0_append_butlast_lchild*: ⟦*well_shaped t*; *well_valued0 as t*⟧ ⟹
well_valued0 (*butlast as* @ [*last as* + *x*]) (*lchild t*)
⟨*proof*⟩

lemma *sum_update_rightmost*: ⟦0 < *i*; *i* ≤ *j*; *length as* = *j*⟧ ⟹
sum as i j + *x* = *sum* (*butlast as* @ [*last as* + *x*]) *i j*
⟨*proof*⟩

lemma *well_valued0_update_rightmost*: ⟦*well_shaped t*; *well_valued0 as t*; *length as* = *r t*⟧ ⟹
well_valued0 (*butlast as* @ [*last as* + *x*]) (*update_rightmost* (λ*a*. *a* + *x*) *t*)
⟨*proof*⟩

lemma *valid_update_rightmost*: ⟦*valid as t*; *length as* = *r t*⟧ ⟹
valid (*butlast as* @ [*last as* + *x*]) (*update_rightmost* (λ*a*. *a* + *x*) *t*)
⟨*proof*⟩

References

[1] D. Basin, F. Klaedtke, and E. Zălinescu. Greedily computing associative aggregations on sliding windows. *Information Processing Letters*, 115(2):186 – 192, 2015.